



```
select hash_value,count(*) "Cost"
  from session_hash
 where sid = &sid
 group by hash_value
 order by 2;
```

**Fig 2.3** - Relative cost of SQL statements

To determine the text for the SQL we can query the v\$sqltext view which stores the statement text in 64 character pieces.

```
select sql_text
  from v$sqltext
 where hash_value = &hash_value
 order by piece;
```

**Fig 2.4** - Querying the text for a SQL statement

To determine the actual resources used by these statements we can query v\$sqlarea

```
select buffer_gets "Consist Reads"
      ,disk_reads "Physi Reads"
      ,sorts, executions
  from v$sqlarea
 where hash_value = &hash_value;
```

**Fig 2.5** - Determining the resource usage for a statement

### 3 The four stages of performance tuning

In the previous section we identified that session's SQL. This was relatively easy because the session id was known. However this might not always be true. In general for most systems first the term *expensive* might need to be defined in terms of resources before *expensive* sessions can be identified. We will use the following methodology for performance tuning -

1. **Categorize the performance degradation (identify bottlenecks)**  
Performance degradation can always be attributed to lack of adequate resources, and the resulting contention for these scarce resources. In this step we identify these resources. Examples of resources are latches, memory, IO. In this step we observe the system and try to define the meaning of expensive in the context of the current system.
2. **Identify the sessions/SQL causing that bottleneck** Any session/statement

making excessive demands (for the resources identified previously as bottlenecks) may be deemed expensive. By filtering the sessions/statements to those making certain demands on the system we can identify the sessions responsible for the load.

3. **Identify and isolate the SQL being executed by the sessions** In the previous two steps we identified the symptoms (e.g. latch contention) and their causes (specific sessions or SQL statements). In this step we further refine the cause to the level of the individual SQL statements causing particular contention. This stage is what we illustrated with the examples in Section 2.
4. **Tune the SQL (fix the problem)** The database executes SQL to accomplish most of its tasks. Consequently fixing the SQL is the most effective way to solve performance problems in many if not most of the cases. Tuning SQL is a vast topic in itself, and we can not do it justice unless the entire paper is devoted to it. Therefore we shall limit the scope of this paper to identifying resource-intensive SQL. We will not discuss tuning tips and techniques.

#### 3.1 Categorize the performance degradation

v\$session\_wait is an excellent view to use as a starting point to find the bottlenecks on the system. The following query will give us a picture of the current waits :

```
select event,sum(decode(
      wait_time, 0,0,1))
      "Previous Waits"
      ,sum(decode(
      wait_time,0,1,0))
      "Currently Waiting"
      ,count(*) "Total Waits"
  from v$session_wait
 group by event
 order by 4;
```

**Fig 3.1** - Sessions currently waiting on the system

Its associated summary views - v\$session\_event and v\$system\_event may be used to get summary information over a period of time. For example We can use the following to get a rough approximation of the

relative waits for various resources since the instance was started

```
select event,total_waits
       from v$system_event
       order by 2;
```

**Fig 3.2** - Summary of waits on the system

If the system is IO bound we can determine which files are getting most IO by querying up the v\$filestat view. By comparing the ratio of the blocks read to the read calls we can actually determine if most of the read calls going to the file are multi-block reads (caused by Full Table Scans or sorts to disk) or single block reads (Indexed Reads).

This is a simple query to determine the ratio for the read calls

```
select b.file#, a.file_name,
       b.phyblkrd "Blks Read"
       ,b.phyblkwrt, b.phyblkrd/
       decode(b.phyrds,0,1,
       b.phyrds) "Ratio"
       from v$dbfiles a
       , v$filestat b
       where a.file_id = b.file#
       order by 5 desc;
```

**Fig 3.3** - Avg no of blocks per physical call

Most well tuned systems tend to be cpu bound. Poorly tuned systems tend to be IO bound (because of the excessive physical reads for each SQL statement). With sufficient amount of memory however we can have a large enough SGA cache most of the required buffers. So a previously IO bound system will now be CPU bound, contending mostly for latches. Consequently the number of buffers accessed tends to be the most expensive resource most of the time.

**Sampling and Snapshotting** - The two methodologies used for problem isolation

The fixed views we use may be broadly classified into two categories

- **Counter views** These views store data as increments from a particular time. e.g. V\$SESS\_IO will continue to increment the number of blocks read. We use these views in two ways -
  1. to obtain statistics since the session (or instance) started

2. to obtain the changes in statistics between two snapshots - the methodology used by bstat/estat.

For example v\$filestat can tell us the overall statistics for file io since the instance started. However if we were to take two snapshots when a performance degradation spike occurs, the differences between the reads in the two snapshots can tell us which files were 'hot' during the spike, and we can get specific results which might otherwise get masked over a longer period of time.

- **Current state views** These views give us a picture of current activity. For example v\$session can tell us the statement currently being executed by a specific session. v\$session\_wait similarly can tell us the event a specific session is currently waiting for. By sampling these views periodically we can classify the more frequently observed events as being more expensive than the less frequently observed events.

### 3.2 Identifying the sessions causing performance bottlenecks

Since practically all the work done in the database involves executing SQL, in most of the cases load may be defined as being proportional to the number of buffers (consistent and disk) accessed. So in most of the cases these two criteria can be used for identifying sessions/SQL as expensive

- Numbers of buffers accessed (disk or consistent)
  - Rate of buffer access
- If disk contention is observed because of heavy IO on some disks then one or more of the following might be the criteria
- sorts to disk (indicated by IO bottleneck on TEMP tablespace files)
  - Excessive Full Table Scans (large number of multi-block reads)
  - Unselective indexes (large number of single block reads)

If latch contention is observed it generally points to problems in some other parts

- Large number of buffers being accessed
- Dynamic SQL causing reparsing for similar statements

Statements running on the system for long periods of time

- Locking issues
- Large numbers of buffers being accessed

### 3.2.1 Statements causing sorts

```
select hash_value,executions,sorts
from v$sqlarea
where sorts > :min_threshold
order by 3 desc;
```

The view v\$sqlarea has a disadvantage that the statistics are not updated until the statement has finished execution. So if the statement is running for the first time these statistics are not going to be accurate. In such a case we need to get this statistic from v\$sesstat and use a combination of sampling and snapshotting techniques similar to the one used below for finding SQL statements causing IO at a high rate.

### 3.2.2 Sessions currently doing a sort

```
select sid
from v$session_wait
where pl in (:file_list)
and event
= 'db file scattered read'
```

*:file\_list* is a list of file#'s for all the files comprising the TEMP tablespace. This will give any session currently reading from the TEMP tablespace (reading the results of a sort). using a multi-block read (event = 'db file scattered read').

### 3.2.3 Statements causing large number of logical/physical reads

```
select hash_value,executions
,buffer_gets,disk_reads
from v$sqlarea
where buffer_gets > &minimum1
or disk_reads > &minimum2
order by buffer_gets +
&cost*disk_reads desc;
```

Here the user can specify the minimum thresholds for the logical and physical reads and the cost of doing a physical read vs a logical read. Since we are selecting from v\$sqlarea we suffer from the same disadvantage as the previous method that if the statement is executing for the first time currently it will not show up on the top. To

work around this we can use the method outlined later which uses IO rates vs total IO.

### 3.2.4 Statements causing Full Table Scans

```
select sql_hash_value hash_value
from v$session
where sid in
(select sid
from v$session_wait
where event =
'db file scattered read'
and wait_time = 0);
```

This query will find all the sessions who are doing a full table scan and waiting for a multi-block read call to return. If we remove the clause **and wait\_Time = 0** it will give all the sessions whose last wait was for a multi-block read. In most of the cases they are still executing the statement causing the full table scans. This is a simple statement to run and quickly find all the 'BAD' statements on the system. But this will not find the worst statements on the system. In a complex query with sufficient number of tables the RBO almost always finds a non-selective index to access the table (which would have been accessed by a Full Table scan for a smaller query). In such a case we read the blocks in one at a time vs doing multi-block reads for a table scan and the query which does not show up on the FT scan queries causes several times the overhead it would have caused if it had caused the FT scan.

### 3.2.5 High IO Rates

If we take two snapshots of the fixed view s a small time interval apart and then display the sessions with the maximum IO done in that interval, then the SQL statement currently running should be responsible for that IO in most cases.

A sample insert for taking the snapshots will be :

```
insert into pm_sess_io
select a.*,sysdate snap_date
, :runid
from v$sess_io a;
```

**Fig 3.2.5.1** - Statement for the snapshot to determine high IO

This will be followed by a second insert and then a report can be generated on the results by the following query

```
select a.sid
       ,(a.block_gets- b.block_gets)/
       ((a.snap_date-b.snap_date)*
        86400) "Blks/sec"
  from   pm_sess_io a,pm_sess_io b
 where  a.sid = b.sid
        and a.runid = b.runid + 1
 order by 2 desc;
```

**Fig 3.2.5.2** - Query for determining high IO sessions from the snapshots

If we do not want to create a temporary table and assume that a session does IO at a constant rate (to find high hitting batch jobs) then we can use Oracle's auditing facility. In this case we set the database initialization parameter

- audit\_trail=db (in init.ora)

If other auditing (within Oracle) is disabled this will only audit connects. Whenever a user logs into the database, a row is inserted with timestamps into the sys.aud\$ table. When the user logs out that row is updated with the logical reads, physical reads and timestamp. By auditing connects we can use a simple query like the one below for finding sessions doing high IO

```

select a.sid
       ,a.block_gets/(sysdate - b.timestamp) "Blks/sec"
  from sys.aud$ b, v$session s, v$sess_io a
 where a.sid = s.sid
       and b.sessionid = s.audsid
 order by 2 desc;

```

**Fig 3.2.5.3** - Using *aud\$* to determine high io sessions

### 3.3 Identifying the SQL being executed by a session

We can use the statements from the previous section for identifying the expensive sessions and sample related SQL statements' hash values. Combining the example **3.2.5.1** with the hash value sampling procedure from Section **2.2**, we can build a simple tool to identify the SQL causing high IO rates. So between the two snapshots we can run this insert periodically :

```

insert into pm_session_hash
       (sid,hash_value,snap_date)
select a.sid, a.sql_hash_value
       ,sysdate
  from v$session a;

```

**Fig 3.3.1** - Sampling v\$session to capture the hash values

Then after the second snapshot we can use this query to identify the high load SQL

```

select s.sid, s.hash_Value "Hash Value" ,count(*) "Cost"
       ,(a.block_gets - b.block_gets)/
       ((a.snap_date-b.snap_date)*86400) "Blks/sec"
  from pm_session_hash s, pm_sess_io a, pm_sess_io b
 where s.sid = a.sid
       and a.sid = b.sid
       and s.snap_date between b.snap_date and a.snap_date
       and a.runid = b.runid + 1
 group by s.hash_Value,(a.block_gets - b.block_gets)/
         ((a.sysdate-b.sysdate)*86400)
 order by 4 desc, 3 desc

```

**Fig 3.3.2** - High IO SQL report

For optimal performance index on pm\_session\_hash(sid,snap\_date) is required in addition to the index on pm\_sess\_io(sid,runid). To save storage space identical hash values may be stored in the same row by using the following procedure

```

procedure sample_session_hash(start_date date) is
cursor c1 is
  select sid,hash_value,sysdate snap_date
  from v$session;
begin
  for chg_tbl in c1 loop
    begin
      update pm_session_hash a
      set hash_count := hash_count + 1
      where a.sid = chg_tbl.sid
      and a.hash_value = chg_tbl.sql_hash_value
      and a.snap_date > start_date;
    exception
      when no_data_found then
        insert into pm_session_hash values
        (chg_tbl.sid,chg_tbl.hash_value,chg_tbl.snap_date,1);
    end;
  end loop;
end;

```

**Fig 3.3.3** - Procedure for sampling SQL executing on the system

The table pm\_session\_hash used throughout section 3.3 can be created by

```

create table pm_session_hash
(sid number, hash_value number,snap_date date,hash_count number);

```

The results may be reported using a query like

```

select s.sid, s.hash_value "Hash Value" ,s.hash_count "Cost"
  ,(a.block_gets - b.block_gets)/((a.snap_date-b.snap_date)*86400)
  "Blks/sec"
from pm_session_hash s,pm_sess_io a,pm_sess_io b
where s.sid = a.sid
  and a.sid = b.sid
  and s.snap_date between b.snap_date and a.snap_date
  and a.runid = b.runid + 1
order by 4 desc,3 desc

```

For performance reasons we need indexes on pm\_session\_hash(sid,hash\_value,snap\_date) and pm\_sess\_io(sid,runid)

## Appendix A - Some Useful Fixed Views

**V\$SESSION** : This view contains session specific information. There is one row in this view for every session currently connected to the instance. Some of the columns of interest of this view are:

- **SID** Session Identifier. This identifier is unique across the view at any given time. These values get reused as one session terminates and another session connects
- **OSUSER** Operating System Userid. This is useful for cases like Oracle Apps where everyone connects as the same Database user but people have different Unix logins.
- **USERNAME** This is the username used for logging into the Database.
- **SQL\_HASH\_VALUE** This is hash value identifier for the statement currently being executed by this session. While executing the statement the session might be waiting for a latch, waiting for the read from the database to return, waiting for the DBWR to flush some buffers to the disk. Since any of these operations pertain: to the execution of the current statement the value in this column will reflect that.

**V\$SQLAREA** This view has statistics specific to the SQL currently in the Shared Pool. There is one row in this view for every shared cursor. Some of the columns of interest are:

- **HASH\_VALUE** The identifier for the Hash Value
- **SQL\_TEXT** First 1000 characters of the statement text
- **EXECUTIONS** Number of times the cursor was executed
- **PARSE\_CALLS** Number of times the cursor was parsed
- **SORTS** Number of times (across executions) the cursor caused sorts
- **BUFFER\_GETS** Total number of logical reads (across executions) made by this cursor
- **DISK\_READS** Number of times this cursor caused a block to fetched to be from the disk

**V\$SQLTEXT** This view contains the complete SQL text for the shared cursors. The text is stored in 64 character pieces. The relevant columns in this view are:

- **HASH\_VALUE** Statement Identifier Hash Value
- **PIECE** Number used for ordering the piece of text
- **SQL\_TEXT** Actual text of the statement in 64 character chunks

**V\$SESSION\_WAIT** This view contains information on the events that sessions are currently waiting for, or the event the session last waited for. It has one row for every session in V\$SESSION. Some of the relevant columns are:

- **SID** Session Identifier for the session which is waiting
- **EVENT** The event this session is waiting for
- **WAIT\_TIME** Amount of time this session actually waited for this event. It is zero if the session is currently waiting for this event.
- **P1, P2, P3** Additional event specific parameters

**V\$FILESTAT** - This view contains system wide IO statistics for every database file since the instance startup. Some of the columns which are relevant are

FILE#	Number of the file - It can be joined to V\$DBFILE to get the name of the file.
PHYRDS	No of physical read calls made to the file
PHYBLKRD	No of blocks read from the file
PHYWRTS	No of physical write calls made to the file
PHYBLKWRT	No of blocks written to the file

**V\$SESS\_IO** - This view contains information on the IO statistics for the sessions. It has one row for every session in V\$SESSION. Some of the relevant columns are

- SID Session Identifier for the session
- BLOCK\_GETS Total number of blocks gotten for the session
- CONSISTENT\_GETS Number of consistent gets for this session
- PHYSICAL\_READS No of physical reads for the session

## **BIBLIOGRAPHY**

- Oracle7 Server Concepts Manual
- Oracle7 Server Tuning
- Oracle7 Server Administrator's Guide

Virag Saksena is a Senior Principal Consultant with Oracle Services' System Performance Group. The team is responsible for building new tools and capabilities like the ones described in this paper for Oracle and its customers. The System Performance Group provides technical architecture services including capacity planning, and performance management services to customers worldwide. His specialization is performance management and application tuning. Virag is based out of

Redwood Shores, California and can be reached at 415.506.5087 or on the internet at **vsaksena@us.oracle.com**. This paper is available via the World-Wide Web at **<http://www.oracle.com/consulting>**  
**<http://www.europa.com/~orapub>**  
and on **[guts.us.oracle.com](http://guts.us.oracle.com)**

## **Acknowledgements**

I would like to acknowledge the support and encouragement I have received from Cary Millsap and Craig Shallahamer. Cary encouraged me to coalesce the performance work I have been doing at client sites into papers like this. Without the constant support and encouragement of Craig, I doubt if I would have gotten this paper together.